Programming Techniques
S.L. Graham, R.L. Rivest Editors

A Data Structure for Manipulating Priority Queues

Jean Vuillemin
Université de Paris-Sud

A data structure is described which can be used for representing a collection of priority queues. The primitive operations are insertion, deletion, union, update, and search for an item of earliest priority.

Key Words and Phrases: data structures, implementation of set operations, priority queues, mergeable heaps, binary trees

CR Categories: 4.34, 5.24, 5.25, 5.32, 8.1

1. Introduction

In order to design correct and efficient algorithms for solving a specific problem, it is often helpful to describe our first approach to a solution in a language close to that in which the problem is formulated. One such language is that of set theory, augmented by primitive set manipulation operations. Once the algorithm is outlined in terms of these set operations, one can then look for data structures most suitable for representing each of the sets involved. This choice depends only upon the collection of primitive operations required for each set. It is thus important to establish a good catalogue of such data structures. A summary of the state of the art on this question can be found in [2]. In this paper, we add to this catalogue a data structure which allows efficient manipulation of priority queues.

A *priority queue* is a set; each element of such a set has a *name*, which is used to uniquely identify the element, and a *label* or *priority* drawn from a totally ordered set. Elements of the priority queue can be thought of as awaiting service, where the item with the smallest label is always to be served next. Ordinary stacks and queues are special cases of priority queues.

A variety of applications directly require using priority queues: job scheduling, discrete simulation languages where labels represent the time at which events are to occur, as well as various sorting problems. These are discussed, for example, in [2, 3, 5, 11, 15, 17, 24]. Priority queues also play a central role in several good algorithms, such as optimal code constructions, Chartre's prime number generator, and Brown's power series multiplication (see [16] and [17]); applications have also been found in numerical analysis algorithms [10, 17, 19] and in graph algorithms for such problems as finding shortest paths [2, 13] and minimum cost spanning tree [2, 4, 25].

Typical applications require primitive operations among the following five: INSERT, DELETE, MIN, UPDATE, and UNION. The operation INSERT (name, label, Q) adds an element to queue Q, while DELETE (name) removes the element having that name. Operation MIN (Q) returns the name of the element in Q having the least label, and UPDATE (name, label) changes the label of the element named. Finally, UNION (Q_1, Q_2, Q_3) merges into Q_3 all elements of Q_1 and Q_2; the sets Q_1 and Q_2 become empty. In what follows, we assume that names are handled in a separate dictionary [2, 17] such as a hash-table or a balanced tree. If deletions are restricted to elements extracted by MIN, such an auxiliary symbol table is not needed.[1]

The heap, a truly elegant data structure discovered by J. W. Williams and R. W. Floyd, handles a sequence of n primitives INSERT, DELETE, and MIN, and runs in $O(n\log n)$ elementary operations using absolutely minimal storage [17]. For applications in which UNION is necessary, more sophisticated data structures have been devised, such as 2–3 trees [2, 17], leftist trees [5, 17], and binary heaps [9].

The data structure we present here handles an arbitrary sequence of n primitives, each drawn from the five described above, in $O(n\log n)$ machine operations and $O(n)$ memory cells. It also allows for an efficient treatment of a large number of updates, which is crucial in connection with spanning tree algorithms: Our data structure provides an implementation (described in [25]) of the Cheriton-Tarjan-Yao [3] minimum cost spanning tree algorithm which is much more straightforward than the original one.

The proposed data structure uses less storage than leftist, AVL, or 2–3 trees; in addition, when the primitive operations are carefully machine coded from the programs given in Section 4, they yield worst case running times which compare favorably with those of their com-

[1] We assume here that indexing through the symbol table is done in constant time.

Author's address: Laboratoire de Recherche en Informatique, Bât. 490, Université de Paris-Sud, Centre d'Orsay 91405—Orsay, France.

Fig. 1. Binary numbering of B_4.

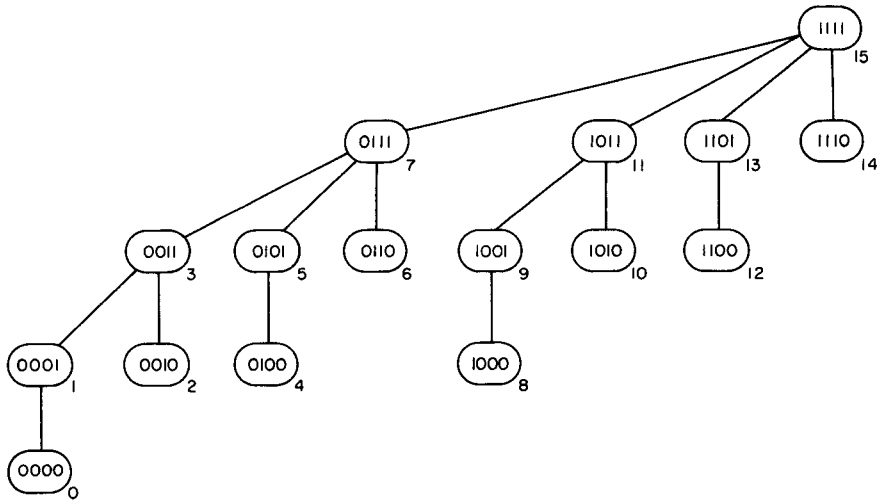


Fig. 2.

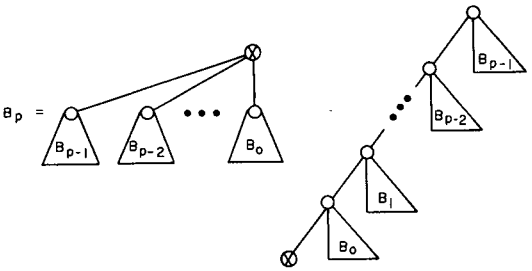
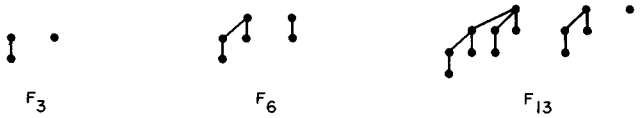


Fig. 3.

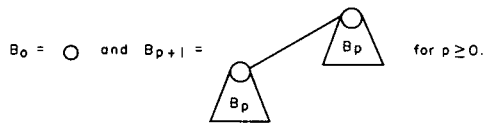


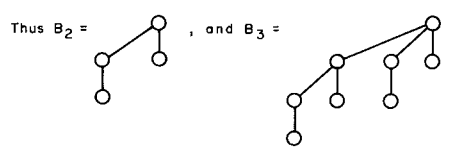
petitors. (A detailed analysis of the algorithms is given by Brown [3].)

Besides these technical advantages, we feel the data structure to be interesting in itself because of its conceptual simplicity and of the connections it establishes among various data manipulation problems.

2. Binomial Trees and Forests

We describe here the underlying combinatorial structure, called *binomial trees*. These are defined inductively by:





for example. In order to discover some of the many combinatorial properties of binomial trees, Knuth [18] suggests that we first label the nodes of a B_p tree in postorder, starting at zero, and then associate with each node the binary representation of its label (Figure 1). From this numbering, it is easy to establish that

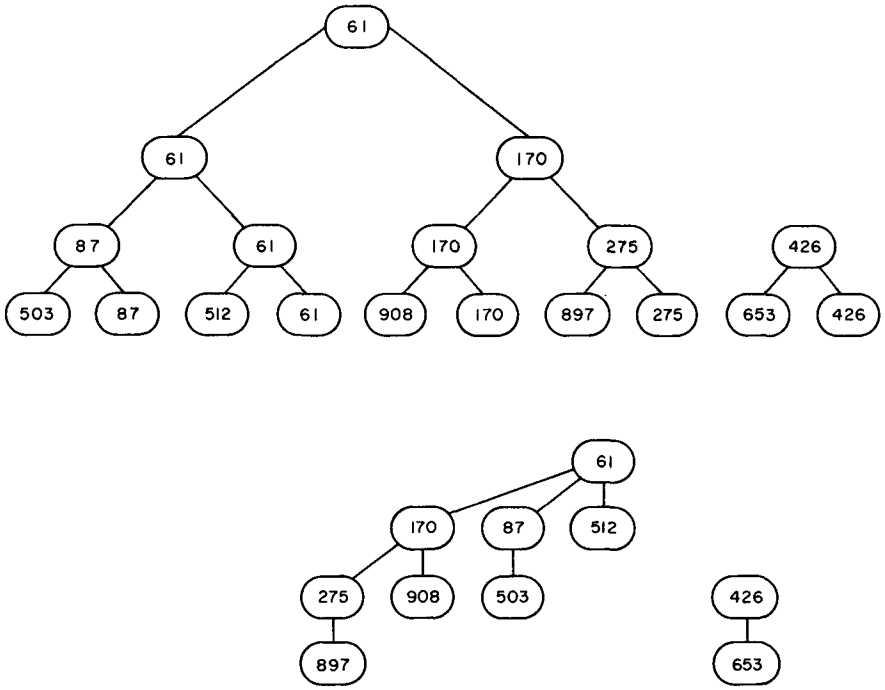
- each B_p has 2^p nodes;
- there are $\binom{p}{k}$ nodes at depth k in B_p which correspond to the various sequences of p bits having exactly k zeros;
- the maximum depth of a node in B_p is p;
- the number of children of a node is equal to the number of 1's following the last 0 in its binary numbering; leaves thus correspond to even numbers;
- in B_p there is exactly one node, the root, having p children; for $0 \leqq k < p$ there are 2^{p-k-1} nodes having k children.

There are many ways of drawing B_p; in particular see Figure 2.

In order to use binomial trees for representing sets whose number n of elements is not always a power of two, we consider the binary decomposition $n = \sum_{i \geq 0} b_i 2^i$, with $b_i \in \{0, 1\}$, of the number n and define *a binomial forest F_n of order n* as a finite collection of binomial trees, one B_i for each 1 in the binary decomposition of n. In symbols, $F_n = \{B_i | i \geq 0, b_i = 1, n = \sum_{i \geq 0} b_i 2^i\}$; we define the ith *component of* F_n to be B_i if $b_i = 1$ and empty otherwise. For example, $(12)_{10} = (1100)_2$; thus $F_{12} = \{B_3, B_2\}$. The first component of F_{12} is empty and its third is B_3. Figure 3 shows some small binomial forests.

Binomial trees and forests appear in various data manipulation problems. They are used by Fisher [6] in the worst case analysis of a simple data structure for manipulating disjoint set unions (see also [18]). They play a crucial role in the linear time median algorithm of Paterson-Pippinger-Schönhage [20]. The Ford-Johnson [8] sorting algorithm can also be nicely described (nonrecursively) with the help of binomial trees; the algorithm first builds a binomial forest, then sorts the partial order thus obtained through a sequence of re-

Fig. 4. The collection {503, 87, 512, 61, 908, 170, 897, 275, 653, 426} of labels represented as: (a) a "perfect" tournament, (b) the same tournament "contracted" into a binomial queue F_{10}.

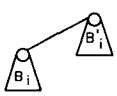


peated "foldings." The data structure presented here can actually be used for implementing the sorting algorithm in time $O(n\log n)$. An efficient machine coding of binary search (see [25]) uses binomial search trees.

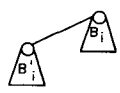
3. Binomial Queues

A priority queue $Q = \{\langle v_1, \lambda_1\rangle, \ldots, \langle v_n, \lambda_n\rangle\}$ consisting of n items is represented by a labeled binomial forest F_n: Each item ⟨name, label⟩ is stored in a different node of F_n subject to the constraint that, if node i is a child of node j in F_n, the labels λ_i and λ_j of the items respectively associated must satisfy $\lambda_i \geq \lambda_j$. This is called the "heap condition" by Knuth [17], and it arises through the natural "contraction" of perfect tournaments as shown in Figures 4(a) and 4(b). Such a labeled binomial forest will be called a *binomial queue.*

We now describe how to perform the UNION of two binomial queues F_n and $F_{n'}$. First consider the special case $n = n' = 2^i$, in which each priority queue is represented by a single labeled binomial tree, say B_i for F_n and B_i' for $F_{n'}$. The resulting forest B_{i+1} = UNION (B_i, B_i') also consists of a single labeled binomial tree with $2^{i+1} = n + n'$ nodes, defined as

$B_{i+1} =$ 

if the label of the root of B_i' is smaller than the corresponding label in B_i and

$B_{i+1} =$ 

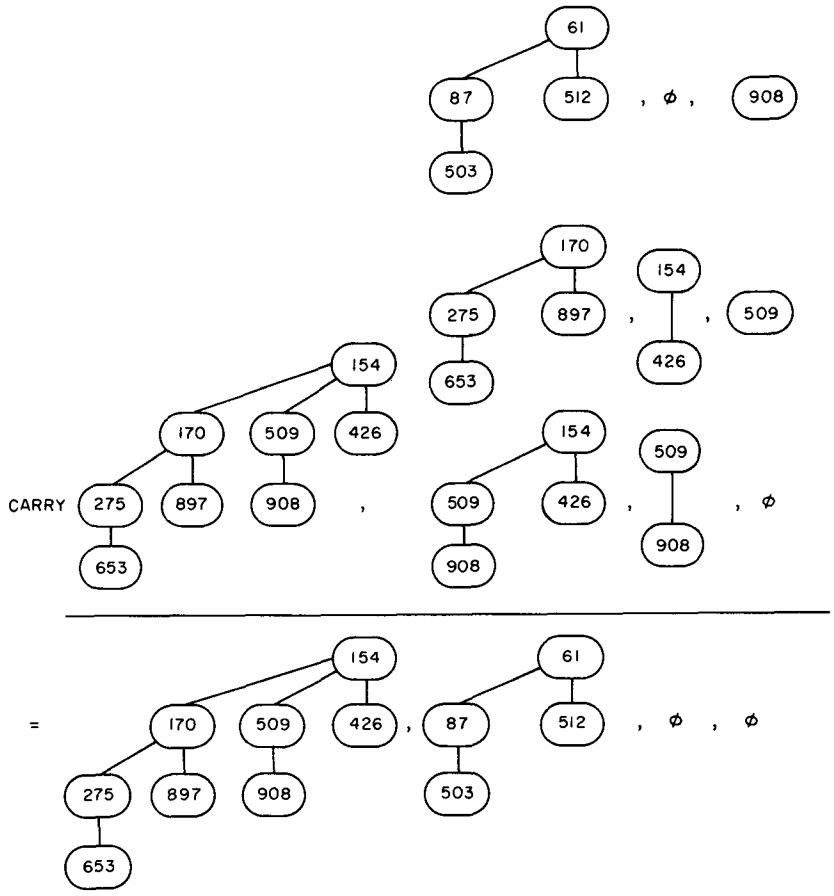
otherwise in order to preserve the heap condition. We refer to this operation as *couplings*, and the general UNION procedure is a sequence of couplings.

For treating the general case where n and n' may be arbitrary, it is convenient to use an analogy with the ordinary scheme for the binary addition of n and n'. The UNION proceeds from low order bits to high order bits; i.e. it treats the binomial trees composing F_n and $F_{n'}$ in order of increasing size.

At each step of the algorithm a carry is propagated; the initial carry is empty, and the carry into the ith step for $i > 0$ is either empty or is a labeled binomial tree B_i. There are three operands at each step of the algorithm which play identical roles; each operand is either empty or a labeled binomial tree B_i. One of the operands is the carry, and the other two are the ith components of F_n and $F_{n'}$ respectively. If all three operands are empty, the ith component of the result UNION (F_n, $F_{n'}$) is empty, as is the carry propagated to the next step. If exactly one operand is nonempty, it constitutes the ith component of the result, and the carry is empty. If two operands are nonempty, they are coupled according to the procedure described earlier in order to constitute the $(i+1)$th carry; the ith component of the result is empty. When all three operands are nonempty, one of them arbitrarily constitutes the ith component of the result, and the remaining two are coupled in order to form the carry. The procedure starts at the 0th step and stops when either F_n or $F_{n'}$ has been exhausted and no carry is propagated any further. The algorithm is pictured with $n = 7$ and $n' = 5$ (Figure 5).

By considering a single item as constituting an F_1 binomial queue, INSERT can be treated as a special case of UNION. A forest F_n is naturally constructed as the

Fig. 5. (a) Binary addition of 7 and 5. (b) Scheme for UNION (F_7, F_5). (c) Actual example.

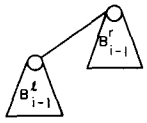


result of a sequence of n INSERT operations. The number of comparisons required by this construction is equal, on the one hand, to the number of carries propagated in the addition $1 + 1 + \cdots + 1$ (n times), and, on the other hand, to the number of edges in the graph of F_n. If $\nu(n)$ denotes the number of ones in the binary decomposition of n, this last number is clearly equal to $n - \nu(n)$. It follows that F_n is constructed in $n - \nu(n)$ comparisons, which is $O(n)$. As for UNION (n, n'), exactly $\nu(n) + \nu(n') - \nu(n + n')$ comparisons are required, which is $O(\log(n + n'))$.

In order to find the minimal label of F_n, we merely have to explore the roots of the binomial trees composing F_n and keep the name of a node having minimal label among these. This involves $\nu(n) - 1$ comparisons, which is $O(\log n)$.

In many applications, DELETE is restricted to extracting the item m found by MIN. Let B_i be the labeled binomial tree in F_n of which m is the root. We first remove B_i from F_n, thus forming a labeled binomial forest F_{n_1} with $n_1 = n - 2^i$. Then the root of B_i is cut. As we can see from Figure 2, what remains is a "complete" forest F_{n_2} with $n_2 = 2^i - 1$. One then calls UNION (F_{n_1}, F_{n_2}) in order to restore F_{n-1} in $O(\log n)$ comparisons again. This procedure is described in the example shown in Figure 6.

If the item m to be removed by DELETE is not the root of one of the components of F_n, the algorithm is slightly more complex. First, we determine the component of F_n in which m lies, say B_i. As before, we remove B_i, thereby forming F_{n_1} with $n_1 = n - 2^i$. We then consider

$B_i =$ 

If m is in B^l_{i-1}, we start constructing $F_{n_2} = \{B^r_{i-1}\}$ and decompose B_{i-1} by the same technique; otherwise, we further decompose B^r_{i-1} and let F_{n_2} include B^l_{i-1}. This continues until m becomes the root of the subtree B_j to be decomposed. It is then "cut" from B_j, thus leaving a complete F_{2^j-1}, which is added to the binomial queue $F_{n_2} = \{B_{i-1}, B_{i-2}, \ldots, B_j\}$ already constructed in order to form a complete F_{2^i-1}. This forest is then merged with F_{n_1} by using the UNION procedure in order to construct the resulting F_{n-1}. This algorithm is illustrated by an example in Figure 7.

As for our last primitive, UPDATE, the obvious way to realize it is to perform DELETE then INSERT in sequence. This requires $O(\log n)$ operations for each UPDATE. If we have to service an arbitrary sequence of primitive operations in which MIN is required less often than UPDATE and DELETE, and UPDATE only causes labels to increase, there is a better way to proceed. The

Fig. 6. (a) Labeled F_{12}. (b) Broken up into an F_4 and an F_7 after removal of 61. (c) F_{11} = UNION (F_4, F_7) reconstituted.

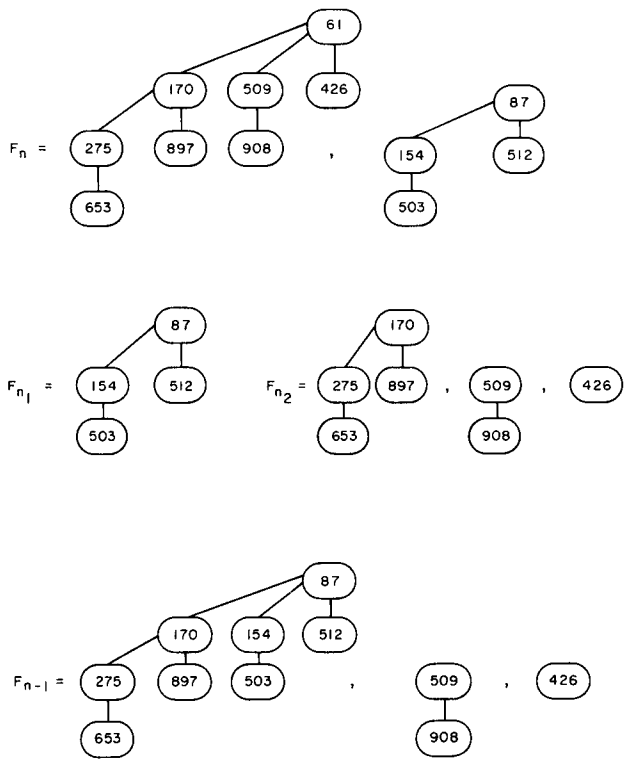
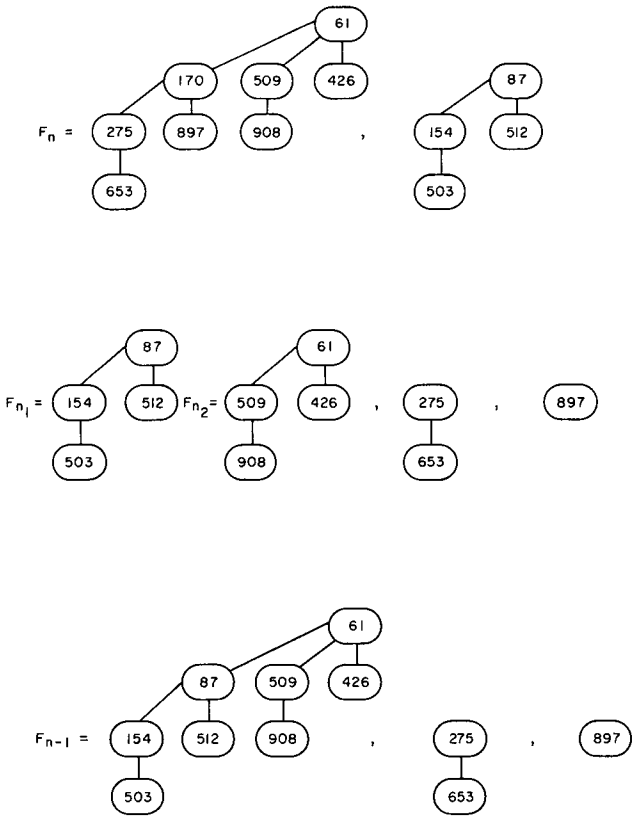


Fig. 7. (a) Labeled F_{12}. (b) Broken up into an F_4 and an F_7 after removal of 170. (c) F_{11} = UNION (F_4, F_7) reconstituted.



procedure UPDATE does not attempt to restore the structure, but merely changes the label and marks the node. The DELETE procedure proceeds in the same way, changing the label to, say, $+\infty$. The algorithm for MIN then becomes more complicated; it explores all binomial trees in the forest F_n until finding unmarked nodes on all paths from the root of the tree to the leaves; this cuts some subtrees and the forest is then reconstructed by merging all these subtrees together with the marked nodes having labels different from $+\infty$. If μ marked nodes are met, it follows (see [25]) from the properties described in Section 2 that the number of trees cut is a most $\mu\log(n/\mu)$. If we consider an arbitrary sequence of n INSERT or UNION, u UPDATE or DELETE, and m MIN, an elementary analysis (see [25] again) shows that such a sequence is treated in at most $O(n\log n) + O(u) + O(u\log(nm/u))$ operations, which is better than the naïve method for $m \leq u$.

Yet another way of treating UPDATE, when labels can only decrease, is described in [14].

4. Implementation of Binomial Forests as Binary Trees

Although the above discussion is an adequate presentation of the priority queue primitives for a very abstract machine model (decision trees for example), it does not indicate how to actually code these algorithms on a digital computer. One still has to solve some problems, the first of which concerns the machine representation of labeled binomial forests.

For this purpose, we represent binomial forests as binary trees through the well known "natural correspondence" described in [16]: Each node has fields *llink* and *rlink* such that llink points to the leftmost child of the node and rlink to the node's right sibling. This leaves some freedom for defining a node's right sibling. For the purpose of the UNION procedure, it is crucial to link small trees to larger siblings on the top level, i.e. for nodes having no parent, and to link large trees to smaller siblings on lower levels, as shown in Figure 8.

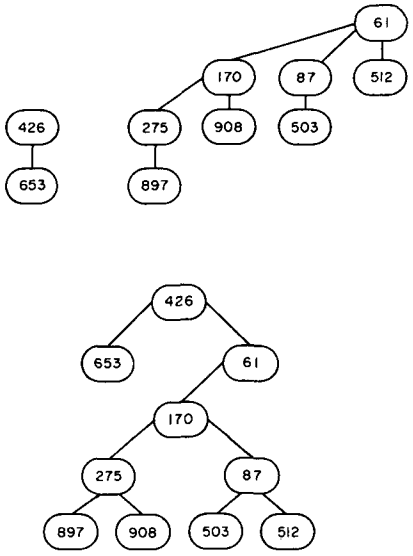
We can now give a formal description of our algorithms in an Algol-like language. Following Knuth [16], we represent a binary tree by three arrays INFO, LLINK, and RLINK containing, respectively, the label, llink, and rlink of each node. The value 0 represents an empty pointer.

We first describe the UNION procedure:

```
proc UNION (R1, N1, R2, N2) → (R3, N3):
  {This procedure merges the two binomial queues F^1 and F^2, yielding
  F^3 for result. Each F^i is represented as a binary tree; R_i is a pointer
  to its root, and N_i represents the number of elements in F^i. The
  initial carry C is zero. Variable RES points to the part of F^3 being
  currently constructed. Location RLINK[0] is used to keep R_3 and thus
  should be available upon calling the procedure.}

  (N3, C, RES) ← (N1 + N2, 0, 0);
  while (min(N1, N2) ≠ 0) ∨ (C ≠ 0)
  do NEXTBIT; (N1, N2) ← (⌊N1/2⌋, ⌊N2/2⌋) od;
  RLINK[RES] ← if N1 ≠ 0 then R1 else R2 fi;
  R3 ← RLINK[0]
endproc UNION.
```

Fig. 8. (a) Binomial queue F_{10}. (b) Same F_{10} as a binary tree. (c) Possible machine representation for this binary tree using arrays; memory cells R and N contain, respectively, a pointer to the root of the tree and its number of elements.



Here NEXTBIT stands for a fragment of program that treats the eight possible cases for the carry and the relevant component of F^1 and F^2:

```
macro NEXTBIT:
  BITC ← if C = 0 then 0 else 1 fi;
  case ⟨N1 mod 2, N2 mod 2, BITC⟩
    000:
    001: (C, RES, RLINK[RES]) ← (0, C, C)
    010: PROPRES(R2)
    011: PROPCARRY(R2)
    100: PROPRES(R1)
    101: PROPCARRY(R1)
    110: CONSTRUCTCARRY
    111: PROPRES(R1); PROPCARRY(R2)
  endcase
endmacro NEXTBIT.
```

We then have to describe the various macro procedures composing NEXTBIT:

```
macro PROPRES(R):
{The number of bits is odd and one of them, namely R, must be added
  to the result.}
  (R, RES, RLINK[RES]) ← (RLINK[R], R, R)
endmacro PROPRES.
macro PROPCARRY(R):
{Bit R must be added to the carry and the carry propagated.}
if INFO[R] < INFO[C]
  then (C, R, LLINK[R], RLINK[C]) ← (R, RLINK[R], C, LLINK[R])
  else (R, LLINK[C], RLINK[R]) ← (RLINK[R], R, LLINK[C])
fi
endmacro PROPCARRY.
```

Note that our multiple assignments are performed in parallel, which can also be achieved sequentially with the help of extra temporary storage locations.

```
macro CONSTRUCTCARRY:
{Bits R1 and R2 are on and a carry must be constructed.}
if INFO[R1] < INFO[R2]
  then (C, R1, R2, LLINK[R1], RLINK[R2]) ← (R1, RLINK[R1],
    RLINK[R2], R2, LLINK[R1])
  else (C, R2, R1, LLINK[R2], RLINK[R1]) ← (R2, RLINK[R2],
    RLINK[R1], R1, LLINK[R2])
fi
endmacro CONSTRUCTCARRY.
```

This completes the description of UNION. We omit the description of the procedure MIN, which is straightforward. (If very frequent uses of MIN are requested, we can keep the value of the minimal label in a special register.)

As for DELETE, we simply treat EXTRACTMIN, where the element having least label is first found then removed. (The DELETE procedure, for which we give no formal code, is very similar. A little complication arises from the necessity of keeping and updating upward parent links.)

```
proc EXTRACTMIN(R, N) → (R', N'):
  {This procedure extracts from the nonempty labeled binomial forest
  F the element having minimal label. The resulting forest F' is
  obtained by merging two forests F¹ and F² which we first construct.}
  (M, PRED, P, S) ← (R, 0, R, RLINK[R]);
  while S ≠ 0
  do if INFO[S] < INFO[M] then (M, PRED) ← (S, P) fi;
    (S, P) ← (RLINK[S], S)
  od {INFO[M] is the minimal label in F.}
  (R1, N1) ← CONSTRUCT(LLINK[M]); N2 ← N − N1 − 1;
  if PRED = 0 then R2 ← RLINK[M] else (R2, RLINK[PRED]) ← (R,
    RLINK[M]) fi;
  (R', N') ← UNION(R1, N1, R2, N2)
endproc EXTRACTMIN.
```

The procedure EXTRACTMIN uses CONSTRUCT, which transforms the binary tree representations of a B_p into the binary tree representation of the complete forest F_{2^p-1} obtained by removing the root of B_p; essentially, this is achieved just by reversing a list.

```
macro CONSTRUCT(RAC) → (RN, N):
if RAC = 0
  then (R, N) ← (0, 0)
  else (R, SUCC, RLINK[RAC], P) ← (RAC, RLINK[RAC], 0, 1);
      while SUCC ≠ 0
      do (R, SUCC, RLINK[SUCC], P) ←
        (SUCC, RLINK[SUCC], R, 2 × P)
      od; N ← 2 × P − 1
fi
endmacro CONSTRUCT.
```

Received June 1976; revised April 1977

References

1. Adel'son-Vel'skii, G.M., and Landis, Y.M. An algorithm for the organisation of information. *Dokl. Akad. Nauk. SSSR 146,* (1962), 263–266.
2. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
3. Brown, M.R. Implementation and analysis of binomial queue algorithms. (to appear in *SICOMP*).
4. Cheriton, D., Tarjan, R.E., Yao, A.C. Finding minimum spanning trees. Res. Rep. Dept. Comptr. Sci. Stanford U., Stanford, Calif., 1975; also *SIAM J. Comptng. 5,* 4 (Dec. 1976), 724–742.
5. Crane, C.A. Linear lists and priority queues as balanced binary trees. Rep. Stan-CS-72-259, Dept. Comptr. Sci., Stanford U., Stanford, Calif., 1972.
6. Fischer, M.J. Efficiency of equivalence algorithms. In *Complexity of Computer Computations,* R.E. Miller and J.W. Thatcher, Eds., Plenum Press, New York, 1972, pp. 158–168.

7. Floyd, R.W. Algorithm 245. Treesort 3. *Comm. ACM 7*, 12 (Dec. 1964), 701.
8. Ford, L.R., and Johnson, S.M. *A tournament problem. Amer. Math. Monthly 66* (1959), 387–389.
9. Françon, J. *Représentation d'une file de priorités par un arbre binaire.* Rapport, Dept. Math., U. de Strasbourg, 1975.
10. Gentleman, W.M. Row elimination for solving sparse linear systems and least squares problems. Dundee Biennial Conf. on Numer. Anal., 1975.
11. Gonnet, G.H. Heaps applied to event driven mechanisms. *Comm. ACM 19*, 7 (July 1976), 417–418.
12. Gonnet, G.H., and Rogers, L.D. An algorithmic and complexity analysis of the heap as a data structure. Res. Rep. CS 75-20, U. of Waterloo, Waterloo, Canada, 1975.
13. Johnson, D.B. Algorithms for shortest paths. Ph.D. Th., Cornell U., Ithaca, N.Y., 1973.
14. Johnson, D.B. Priority queues with update and finding minimum spanning trees. Tech. Rep. 170, Penn. State U., University Park, Pa., 1975.
15. Jonassen, A., and Dahl, O.J. Analysis of an algorithm for priority queue administration. *BIT 15* (1975), 409–422.
16. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968.
17. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
18. Knuth, D.E. *Selected Topics in Computer Science.* Lecture Notes Series, Inst. Math. U. of Oslo, 1973.
19. Malcolm, M.A., and Simpson, R.B. Local versus Global Strategies for Adaptative Quadrature. *ACM Trans. Math. Software, 1*, 2 (June 1975), 129–146.
20. Paterson, M., Pippenger, N., Schönhage, A. *Finding the median.* Theory Comp. Rep. no. 6, U. of Warwick, Coventry, England, 1975.
21. Porter, T., and Simon, I. Random insertion into a priority queue structure. Rep. Stan-Cs-74-460, Dept. Comptr. Sci., Stanford U., Stanford, Calif., 1974.
22. Prim, R. C. Shortest connection networks and some generalizations. *Bell Syst. Tech. J. 36*, 6 (1957), 1389–1401.
23. van Emde Boas, B.P., Kaas, R., and Zijlstra, E. Design and implementation of an efficient priority queue. Math. Centrum Rep. ZW 60/75, Amsterdam, 1975.
24. Vaucher, J.G., and Duval, P. A comparison of simulation event list algorithms. *Comm. ACM 18*, 4 (April 1975), 223–230.
25. Vuillemin, J. Structures de données. Notes de cours de l'ecole d'été, CEA-EDF-IRIA, 1975.
26. Williams, J.W.J. *Algorithm 232.* Heapsort. *Comm. ACM 7*, 6 (June 1964), 347–348.
27. Wyman, F.P. Improved Event-Scanning Mechanisms for Discrete Event Simulation. *Comm. ACM 18*, 6 (June 1975), 350–353.

Programming Techniques

S. L. Graham, R. L. Rivest Editors

Economical Encoding of Commas Between Strings

S. Even
Technician – Israel Institute of Technology
M. Rodeh
IBM Israel Scientific Center

A method for insertion of delimiters between strings without using new symbols is presented. As the lengths of the strings increase, the extra cost, in terms of prolongation, becomes vanishingly small compared to the lengths of the strings.

Key Words and Phrases: string transmission, delimiters, commas, encoding of the integers
CR Categories: 3.57, 3.81, 5.6

1. Introduction

When representing or transmitting a contiguous sequence of strings of varying lengths, a technique is needed for indicating the boundaries of successive strings. The most natural method is to insert a special delimiting character, hereafter called a *comma*, between the strings. With this new character added to the alphabet, the representation of the strings becomes longer by a multiplicative constant greater than 1. By a simple encoding, this effect can be minimized but not completely eliminated.

An alternative method is to place before each string a code word which specifies the string's length. If the length of the strings is bounded, then we can encode it by using a fixed number of alphabet letters. However, if most strings are much shorter than the bound, then an unnecessary waste is introduced. In the absence of such a bound, the method is not applicable at all.

Authors' present addresses: S. Even, Department of Computer Science, Technion, Haifa, Israel; M. Rodeh, IBM Israel Scientific Center, Technion City, Haifa, Israel.